22 File Structure, Disk Scheduling

Readings for this topic: Silberschatz et al., Chapters 11-13; Anderson/Dahlin, Chapter 13.

File: a named sequence of bytes stored on disk. From the OS' standpoint, the file consists of an ordered set of blocks stored on a block device. File system packs bytes into disk blocks, unpacks them again on reading.

Common addressing patterns:

- Sequential: information is processed in order, one piece after the other. This is a common mode: e.g. editor writes out new file, compiler compiles it, etc.
- Random Access: can address any byte in the file directly without passing through its predecessors. E.g. the data set for demand paging, databases.
- Content based: search for blocks with particular values, e.g. hash table, database, dictionary. Not provided by modern operating system.

Modern file systems must address several general problems:

- Disk Management: efficient use of storage space, fast access to files, sharing of storage space and files between several users.
- Naming: how do users select files?
- Protection: which users have access to a file?
- Persistence: information must last safely for long periods of time.
  - across runs of programs.
  - across login sessions.
  - across power failures.
  - across hardware (e.g., disk) failures.
- Concurrency control: several programs/users may read/write a file at roughly the same time.

Disk Management: how should the disk blocks be used to represent the contents of a file? The structure used to describe which sequence of blocks represents a file is called the *file descriptor*. We'll see later that the file descriptors are stored on disk along with the files.

Things to think about in designing a file structure:

- Most files are small.
- Much of the disk is allocated to large files.
- Many of the I/O operations are made to large files.

Thus, per-file storage cost must be low but large files must have good performance.

Contiguous allocation (also called "extent-based"): allocate files as a single "extent", i.e., a sequence of contiguous disk block. Keep a free list of unused extents on the disk. When creating a file, make the user specify its length, allocate all the space at once. Descriptor contains starting block number and size.

- Advantages: fast access, both sequential and random. Simple. Few seeks.
- Drawbacks: external fragmentation makes large files difficult to allocate. Hard to predict needs at file creation time.
- Example: IBM OS/360.

Multiple extents: descriptor is a table of base/size pairs ("extent" pointers). If run out of extent pointers, set BIGFILE flag in descriptor—each table entry now points to an indirect extent (an extent containing base/size pairs of extents containing file data.)

- Advantages: easier to find free extents, can extend file, fast sequential/random access.
- Drawbacks: when disk gets full, lots of small extents, many seeks. External fragmentation.

- Example: DEMOS.

Linked files: each data block contains a link pointer. File descriptor has a pointer to file's first block. Each of file's block points to next block. All free blocks are part of the free list.

- Advantages? Files can be extended, no fragmentation problems. Sequential access is easy: just chase links.
- Drawbacks? Random access is virtually impossible. Lots of seeking, even in sequential access.

Example: TOPS-10, sort of. Alto, sort of.

File Allocation Table (FAT): a table indexed by disk block number. Each entry contains

- -1 if the block is free
- block number of next block, if block is allocated,
- -2 if block is the last block of a file.

File descriptor contains block number of first block.

- Advantages? If FAT is cached in main memory, random access is reasonable efficient.
- Drawbacks? FAT takes lots of space for large disk; must be cached in DRAM for performance.

Example: MS-DOS, OS/2.

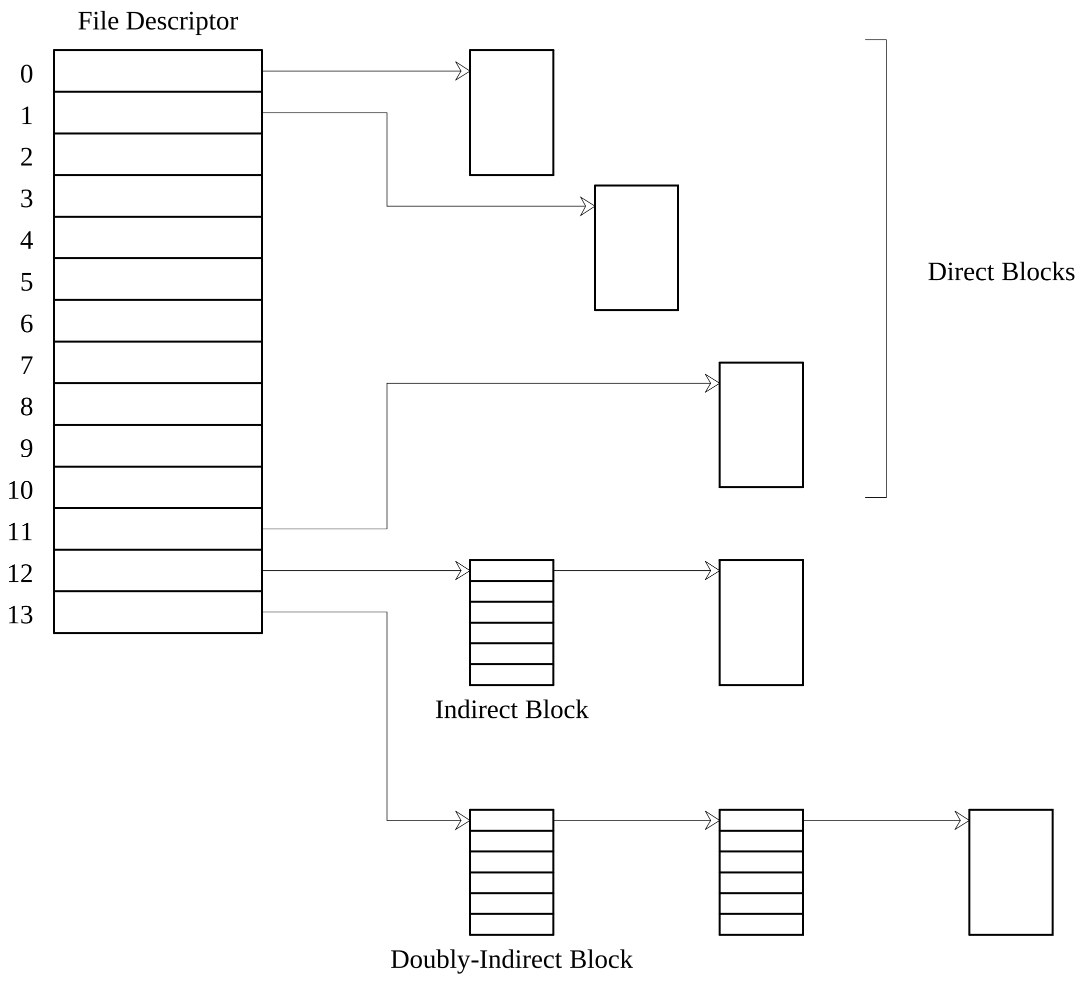
Indexed files: file descriptor has an array of block pointers. File maximum length must be declared when it is created. Allocate an array to hold pointers to all the blocks, but don't allocate the blocks. Then fill in the pointers dynamically using a free list.

- Advantages? Not as much space wasted by overpredicting size; both sequential and random access are easy.
- Drawbacks? Still have to set reasonably tight maximum file size, and there may be lots of seeks.

Multi-level indexed files: Example Unix Fast File System (FFS) solution:

- File descriptor has 15 block pointers. The first 12 point to data blocks, the 13th to an indirect block (contains 1024 more block pointers), the 14th to a doubly-indirect block, and the 15th to a triply-indirect block. Maximum file length is fixed, but large. Indirect blocks aren't allocated until needed. The block size is 4KB.
-
- Advantages: simple, easy to implement, incremental expansion, fast access to small files.
- Potential drawbacks:
  - Indirect mechanism doesn't provide very efficient access to large files: up to 2 descriptor ops for each real operation.
  - Block-by-block organization of free list may mean that file data gets spread around the disk.

UNIX File Descriptors: Multi-Level Indexed



ps2 **Flexible index-tree:** Like FFS, but the index tree depth is configurable on

a per-file basis. The data of a short file can be stored directly in the file descriptor. Also, the index refers to extents rather than individual blocks. Examples: Windows NTFS, Linux ext4, Apple HFS.

Copy-on-write (COW) file systems: Always write new data and metadata into newly allocated blocks. Why?

- large sequential writes are much faster than small random writes
- most reads can be served from a large DRAM page cache
- it matches the characteristics of Flash memory
- cheap disks make it feasible to keep old versions of files
- easy to maintain past snapshots and create clones of the file systems
- easy to roll back (undo) changes to a file and implement atomic updates (see below)

How does it work?

- when writing a file, always allocate a new block
- if needed, allocate (a) new indirect block(s)
- allocate a new file descriptor
  - but then, how do we find the new file descriptor?
  - solution: place all file descriptors into a file
- How do we find the file descriptor of the file descriptor file?
  - we include it in the superblock
- how do we find the superblock? (can't rewrite in place!)

- we allocate a circular array of superblocks
- each superblock containts a version number and checksum
- updates are written into the next available slot
- when booting, we look for the superblock with the highest version number and a valid checksum
- notice this also gives us atomic updates to the filesystem!

Examples: NetApp WAFL, Oracle ZFS, Linux Btrfs

Block cache: a pool of recently-accessed blocks is kept in main memory. If the same blocks are referenced over and over (such as indirect blocks for large files), there's no need to read them from disk repeatedly. This solves the problem of slow access to large files.

Keeping track of free blocks: use a *bit map*.

- Just an array of bits, one per block. 1 means block free, 0 means block allocated. On a 1 TB drive there are 256M 4 KB blocks, so the bit map takes up 32 MBytes. Nowadays, can usually keep entire bit map in memory.
- During allocation, try to allocate next block of file close to previous block of file (extents if possible). If disk isn't full, this will usually work well.
- If disk becomes full, this becomes VERY expensive, and doesn't get much in the way of adjacency. Solution: keep a reserve (e.g. 10% of disk), and don't even tell users about the reserve. Never let the disk get more than 90% full.

Disk scheduling: In servers, it is often the case that there are several disk I/O's requested at the same time, so a scheduler has to decide which to execute first.

First come first served (FIFO, FCFS): may result in a lot of unnecessary seeks under heavy loads, since different requests likely refer to different cylinders of a disk.

Shortest seek time first (SSTF): handle nearest request first. This can reduce seeks and result in greater overall disk efficiency, but some requests may have to wait a long time.

Scan: like an elevator. Move arm back and forth, handling requests as they are passed. This algorithm doesn't get hung up in any one place for very long. It works well under heavy load, but not as well in the middle (about 1/2 the time it won't get the shortest seek).

Anticipatory scheduler (Linux): Take advantage of the fact that an application often issues another (sequential) disk request immediately after the previous completes. To avoid seeks, let the disk go idle for a short while (even if there are disk requests pending for other cylinders) to see if another sequential request arrives.